¡Organiza tu código Javascript de una manera fácil!

Leonardo Souza
06/01/2021

A lo largo de los años en esta industria vital he participado en innumerables proyectos... y eran tantos que los dedos de mis manos serían insuficientes para contarlos. De algunos de ellos estoy orgulloso, de otros no tanto (después de todo, como tú, también produje mucha cosa de calidad dudosa). Aun así, creo en la máxima de que las personas siempre evolucionan, ya que siempre buscan lo mejor dentro de la misión de cada uno.

Organizar el código Javascript para los que se están iniciando no siempre es fácil, porque a juicio del programador novato el lenguaje es el mismo que muchos bloques de código (funciones de lectura) que estaba creando o incluso copiando para solucionar los problemas del día. -día. Este pensamiento, sumado a innumerables malas prácticas, hace que el código hecho con Javascript sea algo caótico.

Entre las principales (malas prácticas), aquí hay algunas que, en mi opinión, entorpecen e irritan a gran parte de los integrantes de un equipo de desarrollo (ah y hace con que la madre del programador sea recordada):

```
falta de sangría

falta de comentarios

variables dispersas
```

```
variables globales

funciones y variables creadas sin propósito
```

Pero el propósito de este artículo no es solo mencionar las partes malas del desarrollo, sino proponer una forma elegante de solucionarlo. Entonces, vayamos a una solución simple para organizar sus códigos hechos con JavaScript.

Funciones nombradas x funciones anónimas

Todos aquí, deben saber qué funciones son **nombradas** y **anónimas** en JavaScript. Si no lo sabe, aquí tiene algunos ejemplos:

```
// funcion nombrada

function common() {

return true;

}
```

Arriba vemos una **función nombrada** y luego una **función anónima**:

```
// función anónima

var anonymous = function () {

return true;

}
```

Mirando los dos códigos, puedes ver una pequeña diferencia, ¿verdad? Los dos cuando se ejecutan devuelven lo mismo (true), pero son estándares sintácticos muy diferentes.

En el primero, tenemos una función definida con el nombre **common**, y es a través de este nombre que será posible invocar esta función, así:

```
common();

// => true
```

En el segundo, tenga en cuenta que la función no tiene nombre, por lo que se tuvo que asignar a la variable **anonymous** para que pueda invocarse. En este caso la llamada sería idéntica al ejemplo anterior:

```
anonymous();

// => true
```

¿Entendiste el concepto? En general, perdón por la ambigüedad, la **función nombrada** se llama así porque **tiene un nombre**. La **función anónima**, a su vez, **no tiene un nombre**, por lo que la única forma de usarla es asignándole algo (una variable, una propiedad en un objeto, etc.). Perdí este tiempo explicando esto, porque creo que valdrá la pena que comprendas lo que vamos a hacer a continuación.

Comenzando la organización

Empecemos los trabajos, **creando una función anónima**:

```
function() {}
```

Si guardas este código e intentas ejecutarlo en tu página, o incluso ejecutarlo en la consola de tu navegador, retomará un **SyntaxError**. Y esto tiene una razón obvia, porque si la función no tiene nombre, ¿cómo la voy a llamar? Entonces, teóricamente, estarías obligado a asignarla a una variable o algo así. En este caso, hagámoslo de otra manera:

```
(function() {})
```

Poniendo este código entre paréntesis, el **SyntaxError** que estuvo rodando hasta entonces desaparecerá. ¿Magia? ¿Magia negra? ¡No! Simplemente los paréntesis agruparon nuestra función, formando una especie de expresión. Este código servirá como un envoltorio

(**wrapper**) para todo lo que haremos en el futuro. Haré una pequeña prueba ahora, poniendo una simpática console.log dentro de nuestra función y, si todo va bien, el resultado será la visualización del texto **"less is more!"**:

```
(function() {

console.log("less is more!");

})
```

Si ejecutaste este código en tu página y echaste un vistazo a la consola, notarás que no hubo retorno. Eh, ¿qué pasó? Resulta que aunque tenemos una función que implementa una sola línea de código, sólo está definida, pero no se ha ejecutado. Para hacer esto, simplemente agregue paréntesis al final de la expresión, así:

```
(function() {

console.log("less is more!");

})();
```

¡Bingo! A partir de ahora, cada vez que se cargue la página, ¡esta función se realizará "automágicamente"! Entonces tenemos la base necesaria para organizar nuestros códigos.

Organizando tu código

Es posible que hayas escuchado que cada variable debe declararse utilizando la palabra clave **var**. No utilices **var**, sale caro, porque una variable definida sin ella se detendrá en **alcance global**, así que para seguir la regla de buenas prácticas siempre la usaremos.

```
(function() {

console.log("less is more!");

var box = {};
```

```
})();
```

En el ejemplo anterior, definí una variable **box** que almacena un **objeto vacío**. La gran ventaja de este enfoque es que **box** no está disponible en ningún propósito que no sea el definido por la función **wrapper** que hicimos. Entonces, si algún tipo inteligente intenta ir a la consola y escribir:

```
console.log(box);

// => ReferenceError: box is not defined
```

Recibirá un bello **ReferenceError** ¡en el medio de la cara! Bueno, ¿verdad? ¡No estropeamos el alcance global y seguimos teniendo privacidad! :)

Una vez que tenemos el objeto **box**, podemos agregar propiedades y métodos a este objeto. Imagínate ahora que necesitamos crear un control muy sencillo, para organizar una fila con los nombres de automóviles. La idea es comenzar con la fila vacía, ir agregando los vehículos y retornar la lista al final. Para esto, usaremos una propiedad llamada **queue** y dos métodos, que llamaremos **addItem** y **getQueue**:

```
(function() {

console.log("less is more!");

// creando el objeto (vacio) box

var box = {};

// agregando la propiedad queue (fila)

box.queue = [];

// agregando los métodos addItem (agregar item)

box.addItem = function() {   };
```

```
// agregando los métodos getQueue (recuperar fila)

box.getQueue = function() {

};

})();
```

Como podemos ver, la propiedad **queue** es un array vacío. Entonces nuestro método **addItem** necesita recibir un vehículo como parámetro y agregarlo a esta fila. A su vez, el método **getQueue** simplemente devuelve una string con los vehículos en la fila, separados por un guión.

```
(function() {

console.log("less is more!");

// creando el objeto (vacio) box

var box = {};

// agregando la propiedad queue (fila)

box.queue = [];

// agregando los métodos addItem (agregar item)

box.addItem = function(car) {

return box.queue.push(car);

};

// agregando los métodos getQueue (recuperar fila)
```

```
box.getQueue = function() {

return box.queue.join(" - ");

};

})();
```

Hecho esto, si ejecutas este código, debes preguntarte: ¿cómo diablos voy a invocar los métodos **addItem** y **getQueue** del objeto **box** que acabo de implementar, ya que no es accesible en el ámbito global? La respuesta es simple, solo haz nuestra función **wrapper** devuelva el objeto **box** (forma simple) o podemos configurar exactamente lo que se devolverá. Vayamos al primer ejemplo:

```
var GLOBALCAR = (function() {

console.log("less is more!");

// creando el objeto (vacio) box

var box = {};

// agregando la propiedad queue (fila)

box.queue = [];

// agregando los métodos addItem (agregar item)

box.addItem = function(car) {

return box.queue.push(car);

};
```

```
// agregando los métodos getQueue (recuperar fila)

box.getQueue = function() {

return box.queue.join(" - ");

};

return box;

})();
```

Como habrás notado, además de devolver el objeto **box**, asigné nuestro **wrapper** a una variable global llamada **GLOBALCAR** (incluso con la palabra clave var, como **GLOBALCAR** no está dentro de una función, es visible en el propósito más amplio de Javascript y se puede acceder desde cualquier lugar). De esa forma, **GLOBALCAR** se convirtió en un objeto. Para probar nuestra implementación podemos ejecutar (te incentivo a que la pruebes a través de la consola):

```
GLOBALCAR;

// => Object {queue: Array[0], addItem: function, getQueue: function}
```

¿Viste eso? **GLOBALCAR** es un objeto que contiene **queue, addItem** y **getQueue**. Entonces podemos ejecutar el método **addItem** para agregar los vehículos a nuestra fila:

```
GLOBALCAR.addItem("Gol");

// => 1

GLOBALCAR.addItem("Palio");

// => 2

GLOBALCAR.addItem("Corsa");
```

```
// => 3
```

Y el método **getQueue** para devolver los ítems que están en la fila:

```
GLOBALCAR.getQueue();

// => "Gol - Palio - Corsa"
```

Genial, ¿verdad? Pero luego, si sabes un poco más sobre JavaScript, te preguntarás ¿por qué es posible acceder a la propiedad **queue** directamente? Esto tiene mucho que ver con cómo devolvemos el objeto **box**. Si desea ocultar esta propiedad, dejando solo los métodos disponibles, puede hacer esto:

```
var GLOBALCAR = (function() {

console.log("less is more!");

// creando el objeto (vacio) box

var box = {};

// agregnando la propiedad queue (fila)

box.queue = [];

// agregando los métodos addItem (agregar item)

box.addItem = function(car) {

return box.queue.push(car);

};

// agregando los métodos getQueue (recuperar fila)

box.getQueue = function() {
```

```
return box.queue.join(" - ");

};

// devolviendo un objeto personalizado (solo con lo necesario)

return {

add: box.addItem,

get: box.getQueue

};

})();
```

Ahora, una nueva prueba en la consola mostrará:

```
GLOBALCAR;

// => Object {add: function, get: function}
```

Tengas en cuenta que nadie más que tú sabrás sobre la existencia de la propiedad **queue**. Lo mismo ocurre con otros métodos que desea implementar, pero no tiene sentido exponerlos.Basado en la nueva implementación, el método **addItem** se convirtió en **add**, y el **getQueue** se convirtió en un simple **get**. Para agregar ítems a nuestra fila, usaremos:

```
GLOBALCAR.add("Gol");

// => 1

GLOBALCAR.add("Palio");

// => 2
```

```
GLOBALCAR.add("Corsa");

// => 3
```

Y para volver a la fila:

```
GLOBALCAR.get();

// => "Gol - Palio - Corsa"
```

Elegante, ¿verdad? Ahora tienes una base sólida para comenzar a organizar tus proyectos en JavaScript. Lo mejor es que pudiste comprender la línea de tiempo del proceso para alcanzar ese objetivo. Disfruta del momento de satisfacción y comparte con la gente que ahora aprendiste a usar un **pattern** para organizar tu código, conocido por ahí como "**Module Pattern**". ¿No es elegante?

En nuestros cursos de JavaScript, vemos aún más buenas prácticas de organización de JavaScript.

Puedes leer también:

- Comprenda la diferencia entre var, let y const en JavaScript
- Cómo organizar el CSS en tu proyecto
- Formatear números en JavaScript

ARTÍCULOS DE TECNOLOGÍA > FRONT END

SEMESTRAL

US$49,90

un solo pago de US$49,90

- 218 cursos
- Videos y actividades 100% en Español
- Certificado de participación
- Estudia las 24 horas, los 7 días de la semana
- Foro y comunidad exclusiva para resolver tus dudas
- Acceso a todo el contenido de la plataforma por 6 meses

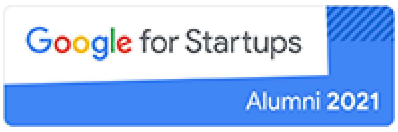
¡QUIERO EMPEZAR A ESTUDIAR!

Paga en moneda local en los siguientes países

ANUAL

ALIADOS

En Alura somos unas de las Scale-Ups seleccionadas por Endeavor, programa de aceleración de las empresas que más crecen en el país.



Fuimos unas de las 7 startups seleccionadas por Google For Startups en participar del programa Growth Academy en 2021

POWERED BY

CURSOS

Cursos de Programación
Lógica de Programación | Java

Cursos de Front End
HTML y CSS | JavaScript | React

Cursos de Data Science
Data Science | Machine Learning | Excel | Base de Datos | Data Visualization | Estadística

Cursos de DevOps
Docker | Linux

Cursos de Innovación y Gestión

Productividad y Calidad de Vida | Transformación Ágil | Marketing Analytics | Liderazgo y Gestión de Equipos | Startups y Emprendimiento